The JML and JUnit Way of Unit Testing and its Implementation

Yoonsik Cheon and Gary T. Leavens

TR #04-02
February 2004

Keywords: Unit testing, automatic test oracle generation, testing tools, runtime assertion checking, formal methods, programming by contract, JML language, JUnit testing framework, Java language.

2000 CR Categories: D.2.1 [*Software Engineering*] Requirements/ Specifications — languages, tools, JML; D.2.2 [*Software Engineering*] Design Tools and Techniques — computer-aided software engineering (CASE); D.2.4 [*Software Engineering*] Software/Program Verification — Assertion checkers, class invariants, formal methods, programming by contract, reliability, tools, validation, JML; D.2.5 [*Software Engineering*] Testing and Debugging — Debugging aids, design, monitors, testing tools, theory, JUnit; D.3.2 [*Programming Languages*] Language Classifications — Object-oriented languages; F.3.1 [*Logics and Meanings of Programs*] Specifying and Verifying and Reasoning about Programs — Assertions, invariants, pre- and post-conditions, specification techniques.

Submitted for publication

An earlier version of this paper appeared in the European Conference on Object-Oriented Programming (ECOOP), Málaga, Spain, June 10-14, 2002. *Lecture Notes in Computer Science*, pp. 231-255, Springer-Veralg. That version was copyright Springer-Verlag, 2002.

Department of Computer Science
226 Atanasoff Hall
Iowa State University
Ames, Iowa 50011-1041, USA

The JML and JUnit Way of Unit Testing and its Implementation

Yoonsik Cheon*
The University of Texas at El Paso

Gary T. Leavens
Iowa State University

February 18, 2004

Abstract

Writing unit test code is labor-intensive, hence it is often not done as an integral part of programming. However, unit testing is a practical approach to increasing the correctness and quality of software; for example, Extreme Programming relies on frequent unit testing.

In this paper we present a new approach that makes writing unit tests easier. It uses a formal specification language's runtime assertion checker to decide whether methods are working correctly; thus code to decide whether tests pass or fail is automatically produced from specifications. Our tool combines this testing code with hand-written test data to execute tests. Therefore, instead of writing testing code, the programmer writes formal specifications (e.g., pre- and postconditions). This makes the programmer's task easier, because specifications are more concise and abstract than the equivalent test code, and hence more readable and maintainable. Furthermore, by using specifications in testing, specification errors are quickly discovered, so the specifications are more likely to provide useful documentation and inputs to other tools. In this paper we describe an implementation using the Java Modeling Language (JML) and the JUnit testing framework, but the approach could be easily implemented with other combinations of formal specification languages and unit testing tools.

1 Introduction

Program testing is an effective and practical way of improving the correctness of software, and thereby improving software quality. It has many benefits when compared to more rigorous methods like formal reasoning and proof, such as simplicity, practicality, cost effectiveness, immediate feedback, understandability, and so on. There is a growing interest in applying program testing to the development process, as reflected by the Extreme Programming (XP) approach [6]. In XP, unit tests are viewed as an integral part of programming. Tests are created before, during, and after the code is written — often emphasized as "code a little, test a little, code a little, and test a little ..." [7]. The philosophy behind this is to use regression tests [33] as a practical means of supporting refactoring.

*An earlier version of this work, with a different implementation, was described in our *ECOOP 2002* paper [15]. The work of both authors was supported in part by a grant from Electronics and Telecommunications Research Institute (ETRI) of South Korea, and by grants CCR-0097907 and CCR-0113181 from the US National Science Foundation.
Authors' addresses: Department of Computer Science, The University of Texas at El Paso, 500 W. University Ave., El Paso, Texas 79902; email: `cheon@cs.utep.edu`; Department of Computer Science, Iowa State University, 226 Atanasoff Hall, Ames, Iowa 50011; email: `leavens@cs.iastate.edu`.

1.1 The Problem

However, writing unit tests is a laborious, tedious, cumbersome, and often difficult task. If the testing code is written at a low level of abstraction, it may be tedious and time-consuming to change it to match changes in the code. One problem is that there may simply be a lot of testing code that has to be examined and revised. Another problem occurs if the testing program refers to details of the representation of an abstract data type; in this case, changing the representation may require changing the testing program.

To avoid these problems, one should automate more of the writing of unit test code. The goal is to make writing testing code easier and more maintainable.

One way to do this is to use a framework that automates some of the details of running tests. An example of such a framework is JUnit [7]. It is a simple yet practical testing framework for Java classes; it encourages the close integration of testing with development by allowing a test suite be built incrementally.

However, even with tools like JUnit, writing unit tests often requires a great deal of effort. Separate testing code must be written and maintained in synchrony with the code under development, because the test class must inherit from the JUnit framework. This test class must be reviewed when the code under test changes, and, if necessary, also revised to reflect the changes. In addition, the test class suffers from the problems described above. The difficulty and expense of writing the test class are exacerbated during development, when the code being tested changes frequently. As a consequence, during development there is pressure to not write testing code and to not test as frequently as might be optimal.

We encountered these problems ourselves in writing Java code. The code we have been writing is part of a tool suite for the Java Modeling Language (JML) [10]. JML is a behavioral interface specification language for Java [37, 36]. In our implementation of these tools, we have been formally documenting the behavior of some of our implementation classes in JML. This enabled us to use JML's runtime assertion checker to help debug our code [9, 13, 14]. In addition, we have been using JUnit as our testing framework. We soon realized that we spent too much time writing test classes and maintaining them. In particular we had to write many query methods to determine test success or failure. We often also had to write code to build expected results for test cases. We also found that refactoring made testing painful; we had to change the test classes to reflect changes in the refactored code. Changing the representation data structures for classes also required us to rewrite code that calculated expected results for test cases.

While writing unit test methods, we soon realized that most often we were translating method pre- and postconditions into the code in corresponding testing methods. The preconditions became the criteria for selecting test inputs, and the postconditions provided the properties to check for test results. That is, we turned the postconditions of methods into code for test oracles. A *test oracle* determines whether or not the results of a test execution are correct [44, 47, 50]. Developing test oracles from postconditions helped avoid dependence of the testing code on the representation data structures, but still required us to write many query methods. In addition, there was no direct connection between the specifications and the test oracles, hence they could easily become inconsistent.

These problems led us to think about ways of testing code that would save us time and effort. We also wanted to have less duplication of effort between the specifications we were writing and the testing code. Finally, we wanted the process to help keep specifications, code, and tests consistent with each other.

1.2 Our Approach

In this paper, we propose a solution to these problems. We describe a simple and effective technique that automates the generation of oracles for testing classes. The conventional way of implementing a test oracle is to compare the test output to some pre-calculated, presumably correct, output [26, 42].

We take a different perspective. Instead of building expected outputs and comparing them to the test outputs, we monitor the specified behavior of the method being tested to decide whether the test passed or failed. This monitoring is done using the formal specification language's runtime assertion checker. We also show how the user can combine hand-written test inputs with these test oracles. Our approach thus combines formal specifications (such as JML) and a unit testing framework (such as JUnit).

Formal interface specifications include class invariants and pre- and postconditions. We assume that these specifications are fairly complete descriptions of the desired behavior. Although the testing process will encourage the user to write better preconditions, the quality of the generated test oracles will depend on the quality of the specification's postconditions. The quality of these postconditions is the user's responsibility, just as the quality of hand-written test oracles would be.

We wrote a tool, `jmlunit`, to automatically generate JUnit test classes from JML specifications. The generated test classes send messages to objects of the Java classes under test; they catch assertion violation exceptions from test cases that pass an initial precondition check. Such assertion violation exceptions are used to decide if the code failed to meet its specification, and hence that the test failed. If the class under test satisfies its interface specification for some particular input values, no such exceptions will be thrown, and that particular test execution succeeds. So the automatically generated test code serves as a test oracle whose behavior is derived from the specified behavior of the target class. (There is one complication which is explained in Section 4.) The user is still responsible for generating test data; however the generated test classes make it easy for the user to add test data.

1.3 Outline

The remainder of this paper is organized as follows. In Section 2 we describe the capabilities our approach assumes from a formal interface specification language and its runtime assertion checker, using JML as an example. In Section 3 we describe the capabilities our approach assumes from a testing framework, using JUnit as an example. In Section 4 we explain our approach in detail; we discuss design issues such as how to decide whether tests fail or not, test fixture setup, and explain the automatic generation of test methods and test classes. In Section 5 we discuss how the user can add test data by hand to the automatically generated test classes. In Section 6 we discuss other issues. In Section 7 we describe related work and we conclude, in Section 8, with a description of our experience, future plans, and the contributions of our work.

2 Assumptions About the Formal Specification Language

Our approach assumes that the formal specification language specifies the interface (i.e., names and types) and behavior (i.e., functionality) of classes and methods. We assume that the language has a way to express class invariants and method specifications consisting of pre- and postconditions.

Our approach can also handle specification of some more advanced features. One such feature is an *intra-condition*, usually written as an `assert` statement. Another is a distinction between normal and exceptional postconditions. A *normal postcondition* describes the behavior of a method when it returns without throwing an exception; an *exceptional postcondition* describes the behavior of a method when it throws an exception.

The Java Modeling Language (JML) [36, 37] is an example of such a formal specification language. JML specifications are tailored to Java, and its assertions are written in a superset of Java's expression language.

Figure 1 shows an example JML specification. As shown, a JML specification is commonly written as annotation comments in a Java source code file. Annotation comments start with `//@` or are enclosed in `/*@` and `@*/`. Within the latter kind of comment, at-signs (`@`) on the beginning

```
public class Person {
    private /*@ spec_public @*/ String name;
    private /*@ spec_public @*/ int weight;
    //@ public invariant name != null && name.length() > 0 && weight >= 0;

    /*@ public behavior
      @   requires n != null &&  name.length() > 0;
      @   assignable name, weight;
      @   ensures n.equals(name) && weight == 0;
      @   signals (Exception e) false;
      @*/
    public Person(String n) { name = n; weight = 0; }

    /*@ public behavior
      @   assignable weight;
      @   ensures kgs >= 0 && weight == \old(weight + kgs);
      @   signals (IllegalArgumentException e) kgs < 0;
      @*/
    public void addKgs(int kgs) { weight += kgs; }

    /*@ public behavior
      @   ensures \result == weight;
      @   signals (Exception e) false;
      @*/
    public /*@ pure @*/ int getWeight() { return weight; }

    /* ... */
}
```

Figure 1: An example JML specification. The implementation of the method `addKgs` contains an error to be revealed in Section 5.3. This error was overlooked in our initial version of this paper, and so is an example of a "real" error.

of lines are ignored. The `spec_public` annotation lets non-public declarations such as private fields `name` and `weight` be considered to be public for specification purposes[1]. The fourth line of the figure gives an example of an invariant, which should be true in each publicly-visible state.

In JML, method specifications precede the corresponding method declarations. Method preconditions start with the keyword `requires`, frame axioms start with the keyword `assignable`, normal postconditions start with the keyword `ensures`, and exceptional postconditions start with the keyword `signals` [28, 36, 37]. The semantics of such a JML specification states that a method's precondition must hold before the method is called. When the precondition holds, the method must terminate and when it does, the appropriate postconditions must hold. If it returns normally, then its normal postcondition must hold in the post-state (i.e., the state just after the body's execution), but if it throws an exception, then the appropriate exceptional postcondition must hold in the post-state. For example, the constructor must return normally when called with a non-`null`, non-empty string `n`. It cannot throw an exception because the corresponding exceptional postcondition is `false`.

JML has lots of syntactic sugar that can be used to highlight various properties for the reader

[1] As in Java, a field specification can have an access modifier determining its visibility. If not specified, the visibility defaults to the visibility of the Java declaration; i.e., without the `spec_public` annotations, both `name` and `weight` could be used only in private specifications.

and to make specifications more concise. For example, one can omit the `requires` clause if the precondition is `true`, as in the specification of `addKgs`. However, we will not discuss these sugars in detail here.

JML follows Eiffel [39, 40] in having special syntax, written `\old(`E`)` to refer to the pre-state value of the expression E, i.e., the value of E just before execution of the body of the method. This is often used in situations like that shown in the normal postcondition of `addKgs`.

For a non-`void` method, such as `getWeight`, `\result` can be used in the normal postcondition to refer to the return value. The method `getWeight` is specified to be *pure*, which means that its execution cannot have any side effects. In JML, only pure methods can be used in assertions.

In addition to pre- and postconditions, one can also specify intra-conditions with specification statements such as the `assert` statement.

2.1 The Runtime Assertion Checker

The basic task of the runtime assertion checker is to execute code in a way that is transparent, unless an assertion is violated. That is, if a method is called and no assertion violations occur, then, except for performance measures (time and space) the behavior of the method is unchanged. In particular, this implies that, as in JML, assertions can be executed without side effects.

We do not assume that the runtime assertion checker can execute all assertions in the specification language. However, only the assertions that it can execute are of interest in this paper.

We assume that the runtime assertion checker has a way of signaling assertion violations to the callers of a method. In practice this is most conveniently done by using exceptions. While any systematic mechanism for indicating assertion violations would do, to avoid circumlocutions, we will assume that exceptions are used in the remainder of this paper.

The runtime assertion checker must have some exceptions that it can use without interference from user programs. These exceptions are thus reserved for use by the runtime assertion checker. We call such exceptions assertion violation exceptions. It is convenient to assume that all such assertion violation exceptions are subtypes of a single assertion violation exception type.

JML's runtime assertion checker can execute a constructive subset of JML assertions, including some forms of quantifiers [9, 13, 14]. In functionality, it is similar to other design by contract tools [34, 39, 40, 48]; such tools could also be used with our approach.

To explain how JML's runtime assertion checker monitors Java code for assertion violations, it is necessary to explain the structure of the instrumented code compiled by the JML compiler. Each Java class and method with associated JML specifications is instrumented, as shown by example in Figure 2. The original method becomes a private method, e.g., `addKgs` becomes `internal$addKgs`. The checker generates a new method, e.g., `addKgs`, to replace it, which calls the original method, `internal$addKgs`, inside a `try` statement.

The generated method first checks the method's precondition and class invariant, if any.[2] If these assertions are not satisfied, this check throws either a `JMLEntryPreconditionError` or a `JMLInvariantError`. After the original method is executed in the `try` block, the normal postcondition is checked, or, if exceptions were thrown, the exceptional postcondition is checked in the third `catch` block. To make assertion checking transparent, the code that checks the exceptional postcondition re-throws the original exception if the exceptional postcondition is satisfied; otherwise, it throws a `JMLNormalPostconditionError` or `JMLExceptionalPostconditionError`. In the `finally` block, the class invariant is checked again. The purpose of the first `catch` block is explained below (see Section 4.1).

Our approach assumes that the runtime assertion checker can distinguish two kinds of precondition violations: *entry precondition violations* and *internal precondition violations*. The former

[2]To handle old expressions (as used in the postcondition of `addKgs`), the instrumented code evaluates each old expression occurring in the postconditions from within the `checkPre$addKgs` method, and binds the resulting value to a private field of the class. The corresponding private field is used when checking postconditions.

```
public void addKgs(int kgs) {
  checkPre$addKgs(kgs); // check precondition
  checkInv(); // check invariant
  boolean rac$ok = true;
  try {
    internal$addKgs(kgs);
    checkPost$addKgs(kgs); // check normal postcondition
  } catch (JMLEntryPreconditionError e) {
    rac$ok = false;
    throw new JMLInternalPreconditionError(e);
  } catch (JMLAssertionError e) {
    rac$ok = false;
    throw e;
  } catch (Throwable e) {
    try { // check exceptional postcondition
      checkExceptionalPost$addKgs(kgs, e);
    } catch (JMLAssertionError e1) {
      rac$ok = false; // an exceptional postcondition violation
      throw e1;
    }
  } finally {
    if (rac$ok) {
      checkInv(); // check invariant
    }
  }
}
```

Figure 2: The top-level of the run-time assertion checker's translation of the `addKgs` method in class `Person`. (Some details have been suppressed.)

refers to violations of preconditions of the method being tested. The latter refers to precondition violations that arise during the execution of the body of the method under test (refer to Section 4.1 for a detailed explanation). Other distinctions among assertion violations are useful in reporting errors to the user, but are not important for our approach.

In JML the assertion violation exceptions are organized into an exception hierarchy as shown in Figure 3. The abstract class `JMLAssertionError` is the ultimate superclass of all assertion violation exceptions. It has several subclasses that correspond to different kinds of assertion violations, such as precondition violations, postcondition violations, invariant violations, and so on. Our assumed entry precondition and internal precondition violations are realized by the types `JMLEntryPreconditionError` and `JMLInternalPreconditionError`. Both are concrete subclasses of the abstract class `JMLPreconditionError`.

3 Assumptions About the Testing Framework

Our approach assumes that separate tests are to be run for each method of each class being tested. We assume that the framework provides *test objects*, which are objects with at least one method that can be called to execute a test. We call such methods of a test object *test methods*. We also assume that one can make composites of test objects, that is, that one can group several test objects into another test object, which, following JUnit's terminology, we will call a *test suite*.

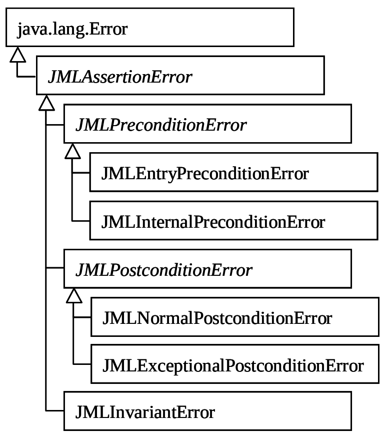
java.lang.Error
JMLAssertionError
JMLPreconditionError
JMLEntryPreconditionError
JMLInternalPreconditionError
JMLPostconditionError
JMLNormalPostconditionError
JMLExceptionalPostconditionError
JMLInvariantError

Figure 3: A part of the exception hierarchy for JML runtime assertion violations.

We further assume that a test method can indicate to the framework whether each test case fails, succeeds, or is meaningless. Following JUnit, a test *failure* means that a bug or other problem was found, *success* means that the code was not found to be incorrect. The outcome of a test is said to be *meaningless* or *rejected* if an entry precondition violation occurs for the test case. Such a test case is inappropriate to test the method, as it is outside the domain of the method under test; details are Section 5.3.

By assuming that the framework provides test objects, which can execute a test, we are also assuming that there is a way to provide test data to test objects.

JUnit is a simple, useful testing framework for Java [7, 30]. In the simplest way to use JUnit, a test object is an object of a subclass of the framework class `TestCase` that is written with several test methods. By convention, each test method has a name beginning with "`test`", returns nothing and takes no arguments. When this is done, the framework can then use Java's introspection API to find all such test methods, and can run them when requested.

Figure 4 is a sample JUnit test class, which is designed to test the class `Person`. A subclass of `TestCase`, such as `PersonTest`, functions both as a class describing a test object and as one describing a test suite. The methods inherited from `TestCase` provides basic facilities for asserting test success or failure and dealing with global test state. `TestCase` also provides facilities for its subclasses to act as composites, since it allows them to contain a number of test methods and aggregate the test objects formed from them into a test suite.

In direct use of JUnit, one uses methods like `assertEquals`, defined in the framework, to write test methods, as in the test method `testAddKgs`. Such methods indicate test success or failure to the framework. For example, when the arguments to `assertEquals` are not equal, the test fails. Another such framework method is `fail`, which directly indicates test failure. JUnit assumes that a test succeeds unless the test method throws an exception or indicates test failure. Thus the only way a test method can indicate success is to return normally.

JUnit thus does not provide a way to indicate that a test execution was meaningless. This is because it is geared toward counting executions of test methods instead of test cases, and because hand-written tests are assumed to be meaningful. However, in our approach we need to extend JUnit to allow the counting of test case executions and to track which test cases were meaningful. We extended the JUnit framework to do this by writing a class `JMLTestRunner`, which tracks the meaningful test cases executed.

JUnit provides two methods to manipulate the global data or state associated with running tests: the `setUp` method creates objects and does any other tasks needed to run a test, and the `tearDown` method undoes otherwise permanent side-effects of tests. For example, the `setUp` method

```
import junit.framework.*;
public class PersonTest extends TestCase {
  private Person p;
  public PersonTest(String name) {
    super(name);
  }
  public void testAddKgs() {
    p.addKgs(10);
    assertEquals(10, p.getWeight());
  }
  protected void setUp() {
    p = new Person("Baby");
  }
  protected void tearDown() {
  }
  public static Test suite() {
    return new TestSuite(PersonTest.class);
  }
  public static void main(String args[]) {
    String[] testCaseName = {PersonTest.class.getName()};
    junit.textui.TestRunner.main(testCaseName);
  }
}
```

Figure 4: A sample JUnit test class.

in Figure 4 creates a new `Person` object, and assigns it to the test fixture variable `p`. Both methods can be omitted if they do nothing. JUnit automatically invokes the `setUp` and `tearDown` methods before and after each test method is executed (respectively).

The static method `suite` creates a test suite, i.e., a collection of test objects created from the test methods found in a subclass of `TestCase`. To run tests, JUnit first obtains a test suite by invoking the method `suite`, and then runs each test in the suite. Besides tests, a test suite can also contain several other test suites. Figure 4 uses Java's reflection facility to create a test suite consisting of all the test methods of class `PersonTest`.

Our goal in automatically generating test oracle code was to preserve as much of this functionality as possible. In particular, it is important to run the user's `setUp` and `tearDown` methods before and after each test. However, instead of writing test methods as methods with names beginning with "`test`", the automatically generated code makes very heavy use of the ability to create test suites from generated test objects.

4 Test Oracle Generation

This section presents the details of our approach to automatically generating JUnit tests from a JML-annotated Java class or interface. We will use the term *type* to mean either a class or an interface.

Our tool, `jmlunit`, produces two Java classes, in two files, for each JML-annotated Java type, C. The first is named C`_JML_TestData`. It is a subclass of the JUnit framework class `TestCase`. The C`_JML_TestData` class supplies the actual test data for the tests; more precisely, in the current tool, users edit this class to supply the test data. The second is the class C`_JML_Test`, which is a test oracle class; it holds the code for running the tests and for deciding test success or failure. For example, running our tool on the class `Person` produces two classes, a skeleton of the class `Person_JML_TestData`, and its subclass, `Person_JML_Test`. However, the file `Person_JML_TestData.java`, would not be generated if it already exists, otherwise the user's test data might be lost.

In the rest of this section, we describe the contents of the class *C*`_JML_Test` generated by the tool. The presentation is bottom-up. Subsection 4.1 describes how test outcomes are determined. Subsection 4.2 describes the methods used to access test data, which allows its superclass, *C*`_JML_TestData`, to supply the test data used. (Users might want to skip to Section 5 after reading Subsection 4.2.) After that we discuss in detail the automatic generation of test objects, test methods, and test suites. Section 5 describes the details of supplying test data in the *C*`_JML_TestData` class.

4.1 Deciding Test Outcomes

In JUnit, a test object implements the interface `Test` and thus has one test method, `run()`, which can be used to run a test consisting of several test methods. In our use of JUnit, each test method runs a single test case and tests a single method or constructor. Conceptually, a *test case* for an instance method consists of a pair of a receiver object (an instance of the class being tested) and a tuple of argument values; for testing static methods and constructors, a test case does not include the receiver object.

The outcome of a call to a method or constructor, M, for a given test case, is determined by whether the runtime assertion checker throws an exception during M's execution, and what kind of exception is thrown. If no exception is thrown, then the test case succeeds (assuming the call returns), because there was no assertion violation, and hence the call must have satisfied its specification.

Similarly, if the call to M for a given test case throws an exception that is not an assertion violation exception, then this also indicates that the call to M succeeded for this test case. Such exceptions are passed along by the runtime assertion checker because it is assumed to be transparent. Hence if the call to M throws such an exception instead of an assertion violation exception, then the call must have satisfied M's specification, specifically, its exceptional postcondition. With JUnit, such exceptions must, however, be caught by the test method, since any such exceptions are interpreted by the framework as signaling test failure. Hence, the test method must catch and ignore all exceptions that are not assertion violation exceptions.[3]

If the call to M for a test case throws an assertion violation exception, however, things become interesting. If the assertion violation exception is not a precondition exception, then the method M is considered to fail that test case.

However, we have to be careful with the treatment of precondition violations. A precondition is an obligation that the client must satisfy; nothing else in the specification is guaranteed if the precondition is violated. Therefore, when a test method calls a method M and M's precondition does not hold, we do not consider that to be a test failure; rather, when M signals a precondition exception, it indicates that the given test input is outside M's domain, and thus is inappropriate for test execution. We call the outcome of such a test execution "meaningless" instead of calling it either a success or failure. On the other hand, precondition violations that arise inside the execution of M should still be considered to be test failures. To do this, we distinguish two kinds of precondition violations that may occur when a test method runs M on a test case, say *tc*:

- The precondition of M fails for *tc*, which indicates, as above, that the test case *tc* is outside M's domain. As noted earlier, this is called an *entry* precondition violation.

[3]Catching and ignoring exceptions does interact with some of JML's syntactic sugars, however. We found that if the user does not specify an exceptional postcondition, it seems best to treat exceptions as if they were assertion violations, instead of using a default that allows them to be treated as test successes. The JML compiler (`jmlc`) has an option (`-U`) to compile files with this behavior.

- A method N called from within M's body signals a precondition violation, which indicates that M's body did not meet N's precondition, and thus that M failed to correctly implement its specification on the test case tc. (Note that if M calls itself recursively, then N may be the same as M.) Such an assertion violation is an *internal* precondition violation.

The JML runtime assertion checker converts the second kind of precondition violation into an internal precondition violation exception. Thus, a test method decides that M fails on a test case tc if M throws an internal precondition violation exception, but rejects the test case tc as meaningless if it throws an entry precondition violation exception. This treatment of precondition exceptions was the main change that we had to make to the JML's existing runtime assertion checker to implement our approach. The treatment of meaningless test case executions is also the only place where we had to extend the JUnit testing framework.

Our implementation of this decision process is shown in the method `runTest` in Figure 5. The user is informed of what is happening during the testing process by the JUnit framework's use of the observer pattern. Most of this is done by the JUnit framework, but the call to the `addMeaningless` method also informs listeners of what is happening during testing (see Section 4.4). The `doCall` method is an abstract method that, when overridden in a particular test object, will call the method being tested with the appropriate arguments for that test case. The test's outcome is recorded as discussed above. There are only two tricky aspects to this. The first is adjusting the assertion failure error object so that the information it contains is of maximum utility to the user. This is accomplished by deleting the stack backtrace that would show only the test harness methods, which happens in the call to the `setStackTrace` method, and adding the original exception as a "cause" of this error. The second tricky aspect that the runtime assertion checker must be turned off when indicating test failure, and turned back on again after that is done. This is needed because the report about the problem can call `toString` methods of the various receiver and argument objects without finding the same violations (for example, invariant violations) that caused the original test failure.

To summarize, the outcome of a test execution is "failure" if an assertion violation exception other than an entry precondition violation is thrown, is "meaningless" if an entry precondition violation is thrown, and "success" otherwise.

4.2 Test Data Access

In this section we describe the interface that a *C*`_JML_Test` class uses to access user-provided test data. Recall that this test data for testing a class C is defined in a class named *C*`_JML_TestData` that is a superclass of *C*`_JML_Test`; thus the interface described in this subsection describes how these classes communicate. In this section we describe only that interface; the ways that the user can implement it in the *C*`_JML_TestData` class are described in Section 5.

Test data provided in the *C*`_JML_TestData` class are used to construct receiver objects and argument objects that form test cases for testing various methods. For example, a test case for the method `addKgs` of class `Person` (see Figure 1) requires one object of type `Person` and one value of type `int`. The first object will be the receiver of the message `addKgs`, and the second will be the argument. In our approach, there is no need to construct expected outputs, because success or failure is determined by observing the runtime assertion checker, not by comparing results to expected outputs.

What interface is used to access the test data needed to build test cases for testing a method M? There are several possibilities:

- *Separate test data.* For each method M to be tested, there is a separate set of test data. This results in a very flexible and customizable configuration. However, defining such fixture variables becomes complicated and requires more work from the user.

```
public void runTest() throws Throwable {
  try {
    // The call being tested!
    doCall();
  }
  catch (JMLEntryPreconditionError e) {
    // meaningless test input
    addMeaningless();
  } catch (JMLAssertionError e) {
    // test failure
    int l = JMLChecker.getLevel();
    JMLChecker.setLevel(JMLOption.NONE);
    try {
      String failmsg = this.failMessage(e);
      AssertionFailedError err = new AssertionFailedError(failmsg);
      err.setStackTrace(new StackTraceElement[]{});
      err.initCause(e);
      result.addFailure(this, err);
    } finally {
      JMLChecker.setLevel(l);
    }
  } catch (Throwable e) {
    // test success
  }
}
```

Figure 5: Template method abstraction of running a test case, showing how failure, success, and meaningless tests are decided.

- *Global test data.* All test methods share the same set of test data. The approach is simple and intuitive, and thus defining the test data requires less work from the user. However, the user has less control in that, because of shared test data, it becomes hard to supply specific test cases for testing specific methods.

- *Combination.* Some combination of the above two approaches, which has a simple and intuitive test fixture configuration, and yet to gives the user more control.

Our earlier work [15] adopted the global test data approach. The rationale was that the more test cases would be the better and the simplicity of use would outweigh the benefit of more control. There would be no harm to run test methods with test cases of other methods (when these are type-compatible). Some of test cases might violate the precondition; however, entry precondition violations are not treated as test failures, and so such test cases cause no problems.

However, our experience showed that more control was necessary, in particular to make it possible to produce clones of test data so that test executions do not interfere with each other.[4] Thus a combination approach is used in the current implementation.

In the combination approach we have adopted, there is an abstract interface between the test driver and the user-supplied C_JML_TestData class that defines the test data.[5] The exact interface used depends on whether the type of data, T, is a reference type or a primitive value

[4]Also, in our earlier implementation, all of the test fixture variables were reinitialized for each call to the method under test, which resulted in very slow testing; most of this reinitialization was unnecessary.

[5]We thank David Cok for pointing out problems with the earlier approach and discussing such extensions.

```
/** Return a new, freshly allocated indefinite iterator that
 *  produces test data of type T. */
public IndefiniteIterator vTIter(String methodName, int loopsThisSurrounds);
```

Figure 6: Methods used for accessing test data of a reference type T by the test oracle class.

type. For a reference type T, the user must define the method shown in Figure 6. The type `IndefiniteIterator` is a special iterator type for supplying test data, and is defined in a package `org.jmlspecs.jmlunit.strategies`. Iterators are convenient for supplying test data because they are easy to combine, filter, etc. The indefinite iterator interface is shown in Figure 7.

For primitive value types, such as `int`, `short`, `float char`, and `boolean`, the interface used is slightly different, since boxing primitive values into objects, and unboxing them later, would be inefficient. Thus, for example, to access data of type `int`, the drivers require the user to define the method shown in Figure 8. The type `intIterator` is an extension of `IndefiniteIterator` that provides direct access to the values of type `int`, using the method `getInt()` instead of `get()`.

To give the user control over what test data is used to test what methods and for which arguments of those methods, the test data access method in Figure 6 has two arguments. The first argument, the string `methodName`, gives the name of the method being tested; this allows different data to be used for different test methods. For example, the user can avoid test data that would cause a time-consuming method to run for a long time by using different test data for just that method. The second argument, `loopsThisSurrounds`, allows the program to distinguish between data generated for different arguments of a method that takes multiple arguments of a given type. It can also be used to help control the number of tests executed. The user that does not want such control simply ignores one or both of these arguments.

To let these test data access methods be shared by all test methods in the test oracle class, we adopt a simple naming convention. Each method's suffix is the name of its type prefixed by the character '`v`', e.g., `vint` for type `int`.[6] If C is a class to be tested and $T_1, T_2, \ldots, T_n$ are the set of other formal parameter types of all the methods to be tested in the class C, then the test data access methods need to be provided for each distinct type in C, T_1, …, T_n.

The test cases used are, in general, the cross product of the set of test data supplied for each type. However, to give the user more control over the test cases, the factory methods shown in Figure 9 is called to create the test suites that are used both to accumulate all the tests and the tests for each method M. The user can override the `emptyTestSuiteFor` method to provide a test suite that, for example, limits the number of tests run for each method to 10. By overriding the `overallTestSuite` method, one can, for example, use only 70% of the available tests. Users not wanting such control can simply use the defaults provided, as shown in Figure 9, which use all available tests.

For example, for the class `Person`, the programmer will have to define the test data access methods shown in Figure 10. These are needed because `Person`'s methods have receivers of type `Person`, and take arguments of types `String` (in the constructor) and `int` (in the constructor and in the method `addKgs`).

The set of test cases for the method `addKgs` is thus all tuples of the form (r, i) where r is a non-null object returned by `vPersonIter("addKgs",1)`, and i is an element returned by `vintIter("addKgs", 1-1)`. The second argument, `loopsThisSurrounds`, is used to distinguish between data generated for different arguments of a method that takes multiple arguments of a given type. Similarly, the set of test cases for the method `getWeight` consists of all tuples of the form (r) consisting of a non-null object r returned by `vPersonIter("getWeight",0)`.

[6]For an array type, the character `$` is used to denote its dimension, e.g., `vint_$_$Iter` for `int[][]`. Also to avoid name clashes, the `jmlunit` tool uses fully qualified names for reference types; for example, instead of `vStringIter`, the actual declaration would be for a method named `vjava_lang_StringIter`. To save space, we do not show the fully qualified names.

```
package org.jmlspecs.jmlunit.strategies;
import java.util.NoSuchElementException;
/** Indefinite iterators. These can also be thought of as cursors
 * that point into a sequence. */
public interface IndefiniteIterator extends Cloneable {
  /** Is this iterator at its end?  I.e., would get() not work? */
  /*@ pure @*/ boolean atEnd();

  /** Return the current element in this iteration.  This method may
   * be called multiple times, and does not advance the state of the
   * iterator when it is called.  The idea is to permit several
   * similar copies to be returned (e.g., clones) each time it is
   * called.
   * @throws NoSuchElementException, when atEnd() is true.
   */
  /*@ public behavior
    @   assignable \nothing;
    @   ensures_redundantly atEnd() == \old(atEnd());
    @   signals (Exception e) e instanceof NoSuchElementException && atEnd();
    @*/
  /*@ pure @*/ Object get();

  /** Advance the state of this iteration to the next position.
   * Note that this never throws an exception. */
  /*@ public normal_behavior
    @   assignable objectState;
    @*/
  void advance();

  /** Return a new iterator with the same state as this. */
  /*@ also
    @  public normal_behavior
    @    assignable \nothing;
    @    ensures \fresh(\result) && \result instanceof IndefiniteIterator;
    @*/
  /*@ pure @*/ Object clone();
}
```

Figure 7: The indefinite iterator interface.

```
  /** Return a new, freshly allocated indefinite iterator that
   *  produces test data of type int. */
  public intIterator vintIter(String methodName, int loopsThisSurrounds);
```

Figure 8: Methods used for accessing test data of the primitive value type `int` by the test oracle class.

```
/** Return the overall test suite for testing; the result
 * holds every test that will be run.
 */
public TestSuite overallTestSuite() {
  return new TestSuite("Overall test suite");
}

/** Return an empty test suite for accumulating tests for the
 * named method.
 */
public TestSuite emptyTestSuiteFor(String methodName) {
  return new TestSuite(methodName);
}
```

Figure 9: Factory methods used for producing test suites, with their default implementations.

```
public IndefiniteIterator vPersonIter(String methodName, int loopsThisSurrounds);
public IndefiniteIterator vStringIter(String methodName, int loopsThisSurrounds);
public IntIterator vintIter(String methodName, int loopsThisSurrounds);
```

Figure 10: Test data access methods the programmer must define for testing the class `Person`.

4.3 Test Objects

For each test case, a test object is automatically created to hold the test case's data and run the corresponding test method. These test objects are created in the C`_JML_Test` class. Their creation starts with a call to the `suite()` method of that class. As shown in Figure 11, this method accumulates test objects into the test suite created by the `overallTestSuite()` factory method described earlier. The `addTestSuite()` method is used to add a test suite consisting of all the custom testing methods, named `test`X, that the user may have defined.[7] Then test suites consisting of test objects for testing each method are also added to this test suite. It is the generation of these test suites, one for each method M to be tested, that will concern us in the remainder of this subsection.

To describe our implementation, let C be a Java type annotated with a JML specification and C`_JML_Test` the JUnit test class generated from the type C. Recall that C`_JML_Test` is a subclass of C`_JML_TestData`. For each instance (i.e., non-`static`) method of the type C with a signature involving only reference types (where $n \geq 0$ and $m \geq 0$):

T M`(`A_1 a_1`,...,` A_n a_n`) throws` E_1`,...,` E_m`;`

the method shown in Figure 13 is used to generate the test objects and add them to the corresponding test suite. It appears, once for each such method M, in the body of the test class.

The nested `for` loops in Figure 13 loop over all test cases for method M. Each loop iterates over all test inputs that correspond to the receiver or a formal parameter. The iterators used in the loop are created by calling the `v`C`Iter` or `v`A_i`Iter` methods. The `v`C`Iter` iterator, which is used to provide the receivers, is filtered to take out null elements, using the class `NonNullIteratorDecorator`.

Each iterator used is checked to see that it is not already at its end. This is done to ensure that there is some testing to be done for the method M. If one of the loops had no test data, then no testing would be done for that method.

[7]Such methods may be written by the user as usual in JUnit.

```
public static Test suite() {
  C_JML_Test testobj = new C_JML_Test("C_JML_Test");
  TestSuite testsuite = testobj.overallTestSuite();
  // Add instances of Test found by the reflection mechanism.
  testsuite.addTestSuite(C_JML_Test.class);
  testobj.addTestSuiteForEachMethod(testsuite);
  return testsuite;
}

public void addTestSuiteForEachMethod(TestSuite overallTestSuite$) {
  ⟨ A copy of the code from Figure 12 for each method and constructor M ⟩
}
```

Figure 11: The method `suite` and `addTestSuiteForEachMethod`. These are part of the C`_JML_Test` class used to test type C.

```
this.addTestSuiteFor$TestM(overallTestSuite$);
```

Figure 12: The call to the `addTestSuiteFor$Test`M for adding tests for the method or constructor M.

The test data in the test case, is passed to the constructor of a specially created, nested subclass, `Test`M;[8] this constructor stores the data away for the call to the test method. The test object thus created is added to the method's test suite. If the method's test suite is full, then this exception causes the loops to terminate early, saving whatever work is needed to generate the test data. At the end of Figure 13, the method's test suite is added to the overall test suite.

Figure 13 shows the code generated when all of the argument types, A_i are reference types. If some of the types A_i are primitive value types, then the appropriate iterator type is used instead of `IndefiniteIterator` in Figure 13, and instead of calling `get()`, the argument is obtained by calling the `get`A_i`()` method appropriate for the type A_i.

If the method being tested is a static method, then the outermost `while` loop in Figure 13 is omitted, since there is no need for a receiver object. Constructors (for non-abstract classes) are tested in a similar way.

By default, the `jmlunit` tool only generates test methods for public methods in the class or interface being tested. It is impossible to test private methods from outside the class, but users can tell the tool whether they would like to also test protected and package visible methods. Normally, the tool only tests methods for which there is code written in the class being tested. However, the tool also has an option to test inherited methods that are not overridden. This is useful because in JML one can add specifications to a method without writing code for it; this can happen, for example, if one inherits specifications through an interface for a method inherited from a superclass. Also, test methods are not generated for a `static public void` method named `main`; testing such a main method seems inappropriate for unit testing.

Figure 14 is an example of the code used to generate test objects for the testing of the `addKgs` method of the class `Person`.

[8]In the implementation, the initial character of M is capitalized in `Test`M, e.g., `TestAddKgs`; this follows the Java conventions for class names.

```
/** Add tests for the M method to the overall test suite. */
private void addTestSuiteFor$TestM(TestSuite overallTestSuite$) {
  methodTests$ = emptyTestSuiteFor("M");
  IndefiniteIterator receivers$iter
    = new NonNullIteratorDecorator(vCIter("M",1));
  check_has_data(receivers$iter,"receiversIter(\"M\",n)");
  try {
    while (!receivers$iter.atEnd()) {
      IndefiniteIterator vA_1$1$iter = vA_1Iter("M",n-1);
      check_has_data(vA_1$1$iter,"vA_1Iter(\"M\",n-1)");
      while (!vA_1$1$iter.atEnd()) {
        ...
          IndefiniteIterator vA_n$n$iter = vA_nIter("M",n-n);
          check_has_data(vA_n$n$iter,"vA_nIter(\"M\",n-n)");
          while (!vA_n$n$iter.atEnd()) {
            final C receiver$ = (C) receivers$iter.get();
            final A_1 a_1 = vA_1$1$iter.get();
            ...
            final A_n a_n = vA_n$n$iter.get();
            methodTests$.addTest(new TestM(receiver$,a_1,...,a_n));
            vA_n$n$iter.advance();
          }
          ...
          vA_1$1$iter.advance();
        }
        receivers$iter.advance();
     }
  } catch (TestSuiteFullException e$) {
     // methodTests$ doesn't want more tests
  }
  overallTestSuite$.addTest(methodTests$);
}
```

Figure 13: Code that generates test objects for method M.

4.4 Test Object Classes

For each method M of class C to be tested, the test driver class C`_JML_Test` contains a nested class `Test`M that describes the test objects that hold the data for a test case and the test method that calls M for that test case. Suppose again that M is an instance method of class C with a signature involving types:

T M`(`A_1 a_1`,...,` A_n a_n`) throws` E_1`,...,` E_m`;`

Then the code shown in Figure 15 describes the class `Test`M used to hold the data and test method for testing M. The constructor shown in Figure 15 simply records its arguments in the object's fields; it is used in Figure 13. The `doCall` method calls method M with these arguments; it is used in Figure 5. The other method, `fail`, is used in formatting the output for the user; it is tailored to M in the sense that it displays the test case's data in the message passed to the JUnit framework's `fail` method.

```
/** Add tests for the addKgs method to the overall test suite. */
private void addTestSuiteFor$TestAddKgs(TestSuite overallTestSuite$) {
  methodTests$ = this.emptyTestSuiteFor("addKgs");
  IndefiniteIterator receivers$iter
    = new NonNullIteratorDecorator(vPersonIter("addKgs", 1));
  check_has_data(receivers$iter,
              "new NonNullIteratorDecorator(vPersonIter(\"addKgs\",1))");
  try {
    while (!receivers$iter.atEnd()) {
      IntIterator vint$1$iter = vintIter("addKgs", 1-1);
      check_has_data(vint$1$iter,"vintIter(\"addKgs\", 1-1)");
      while (!vint$1$iter.atEnd()) {
        final Person receiver$ = (Person) receivers$iter.get();
        final int kgs = vint$1$iter.getInt();
        methodTests$.addTest(new TestAddKgs(receiver$, kgs));
        vint$1$iter.advance();
      }
      receivers$iter.advance();
    }
  } catch (TestSuiteFullException e$) {
    // methodTests$ doesn't want more tests
  }
  overallTestSuite$.addTest(methodTests$);
}
```

Figure 14: Test object generation code for the method `addKgs` of the class `Person`.

For example, the nested class `TestAddKgs` that describes the test objects used for testing the method `addKgs` is shown in Figure 16.

Each of these nested classes named `Test`M in C`_JML_Test` is a subclass of the class `OneTest`. The class `OneTest` is also nested within the test driver class, C`_JML_Test`. The class `OneTest` is an abstract class that is a subclass of the class C`_JML_Test` in which it is nested. It holds common code for the test object classes. Figure 17 shows the code of `OneTest` that is generated inside `Person_JML_Test`. (The code for the general case is the same, except that the superclass of `OneTest` is, in general, C`_JML_Test`.)

The code for the method `runTest` in `OneTest` was given already, in Figure 5. The override of the `run` method is only needed to squirrel away the test result object into a field of `OneTest`, so that it can be used in the `runTest` method. Most of `run`'s work is done by the inherited method which is super-called from within `run`. This `run` method of the JUnit framework is the one that calls back down to the `runTest` method to actually run the test. The private method `addMeaningless` in Figure 17 is called by `runTest`, and adds extra information to the result object if it is a `JMLTestResult` object. The extra information containing the meaningfulness of test cases becomes available if the tests are run with the JML extension of the JUnit's test runner class (see Section 5.3 for an example).

4.5 Test Driver Classes

The form of the test driver class, C`_JML_Test`, is shown in outline in Figure 18.

As mentioned above, C`_JML_Test`, is a subclass of C`_JML_TestData`, which in turn is a subclass of the JUnit framework's class `TestCase`. Since the nested class `OneTest` inherits from C`_JML_Test`, and since the nested `Test`M classes all inherit from `OneTest`, this allows the test objects (of type `Test`M

```
protected static class Test$M$ extends OneTest {
  /** The receiver */
  private $C$ receiver$;
  /** Argument $a_1$ */
  private $A_1$ $a_1$;
  ...
  /** Argument $a_n$ */
  private $A_n$ $a_n$;

  /** Initialize this Test$M$ instance. */
  public Test$M$($C$ receiver$, $A_1$ $a_1$, ..., $A_n$ $a_n$) {
    super("$M$");
    this.receiver$ = receiver$;
    this.$a_1$ = $a_1$;
    ...
    this.$a_n$ = $a_n$;
  }

  protected void doCall() throws Throwable {
    receiver$.$M$($a_1$, ..., $a_n$);
  }

  protected void failMessage(JMLAssertionError e$) {
    String msg = "\n\tMethod '$M$' applied to";
    msg += "\n\tReceiver: " + this.receiver$;
    msg += "\n\tArgument $a_1$: " + this.$a_1$;
    ...
    msg += "\n\tArgument $a_n$: " + this.$a_n$;
    return msg;
  }
}
```

Figure 15: Code that generates test objects for method M.

for some M) to inherit the `setUp` and `tearDown` methods written by the user in C`_JML_TestData`. This complicated inheritance structure is designed just for that purpose, since `setUp` and `tearDown` play a prominent role in JUnit testing.

The `package` definition for C`_JML_Test` is copied verbatim from the type C. As a result, the generated test class will reside in the same package. (This can be changed, however, with an option to the `jmlunit` tool.) Having the test class in the same package as C allows the test class to access C's package-visibility members during testing. In this paper we act as if the packages in our implementation were imported into the code, but this is just for the sake of brevity; the generated code uses fully-qualified names.

In addition to the static `suite` and `main` methods described in Section 4.3 above, the C`_JML_Test`, has one other prominent method. As shown in Figure 18, this is the method `test$IsRACCompiled`. This method checks that the code under test was compiled by the JML compiler, thus enabling runtime assertion checking. This test prevents the user from thinking that the other tests have passed when in fact no testing was done because no assertions were checked.

```
protected static class TestAddKgs extends OneTest {
  /** The receiver */
  private Person receiver$;
  /** Argument kgs */
  private int kgs;

  /** Initialize this TestAddKgs instance. */
  public TestAddKgs(Person receiver$, int kgs) {
    super("addKgs");
    this.receiver$ = receiver$;
    this.kgs = kgs;
  }

  protected void doCall() throws Throwable {
    receiver$.addKgs(kgs);
  }

  protected void failMessage(JMLAssertionError e$) {
    String msg = "\n\tMethod 'addKgs' applied to";
    msg += "\n\tReceiver: " + this.receiver$;
    msg += "\n\tArgument kgs: " + this.kgs;
    return msg;
  }
}
```

Figure 16: Class describing test objects for testing the method `addKgs` of the class `Person`.

5 Supplying Test Data and Running Test Cases

To provide test data for testing a type C, the user must write code into the class C`_JML_TestData`. The test data is provided by implementing methods described in Section 4.2 above; we will give more details and examples of how this is done in this section.

Besides providing test data, the user can tune the testing by adding hand-written test methods to the C`_JML_TestData` class. The framework collects all test methods whose names begin with "`test`", putting them into the test suite as described in Section 4.3.

The `jmlunit` tool produces a skeleton subclass of the test data class automatically, if one does not exist. For example, if the file `Person_JML_TestData.java` does not exist, then when the tool is run on the type `Person`, it produces a test data class skeleton shown in outline in Figure 19. The methods of Figure 9 are the factory methods that the user can change to customize the test suites used, as described in Section 4.3.

5.1 Supplying Test Data

The actual test inputs are supplied by overriding the test data access methods declared in the test oracle class for each type of data (see Figures 6 and 8). Test data are ultimately supplied as indefinite iterator objects (see Figure 7). How these indefinite iterator objects are constructed is ultimately up to the programmer of the test data class; however the tool generates skeletons corresponding to a few standard strategies for providing access to test data.

```
  /** A JUnit test object that can run a single test method. */
  protected static abstract class OneTest extends Person_JML_Test {
    protected TestResult result;

    public OneTest(String name) { super(name); }

    public void run(TestResult result) {
      this.result = result;
      super.run(result);
    }

    ⟨ The runTest() method from Figure 5 ⟩

    protected abstract void doCall() throws Throwable;
    protected abstract void failMessage(JMLAssertionError e);

    private void addMeaningless() {
      if (result instanceof JMLTestResult) {
         ((JMLTestResult)result).addMeaningless(this);
      }
    }
  }
```

Figure 17: The nested class OneTest inside Person_JML_Test.

```
⟨ Package directive from C ⟩
import junit.framework.*;
import org.jmlspecs.jmlunit.strategies.*;

public class C_JML_Test extends C_JML_TestData {

  public C_JML_Test(String name) { super(name); }

  public static void main(String[] args) {
    org.jmlspecs.jmlunit.JMLTestRunner.run(suite());
  }

  ⟨ The suite method from Figure 11 ⟩
  ⟨ The class TestM from Figure 15 for each method M ⟩
  ⟨ The superclass OneTest, as in Figure 17 ⟩

  public void test$IsRACCompiled() {
    junit.framework.Assert.assertTrue("code for class C"
      + " was not compiled with jmlc"
      + " so no assertions will be checked!",
    org.jmlspecs.jmlrac.runtime.JMLChecker.isRACCompiled(C.class) );
  }
}
```

Figure 18: Outline of the *C*_JML_Test class used to test type *C*.

```
import junit.framework.*;
import org.jmlspecs.jmlunit.strategies.*;

public abstract class Person_JML_TestData extends TestCase {
  public Person_JML_TestData(String name) {
    super(name);
  }
  ⟨ Factory methods from Figure 9 ⟩
  ⟨ Test data access methods shown in Figures 21 to 24 ⟩
}
```

Figure 19: The test data for the class `Person`, `Person_JML_TestData`.

```
package org.jmlspecs.jmlunit.strategies;

/** Strategies for providing test data.  These simply consist of a
 * method iterator(), which can be called to return a new
 * IndefiniteIterator object. */
public interface StrategyType {

  /** Compute a fresh indefinite iterator, which can be used to
   * provide test data of some reference type. */
  //@ assignable objectState;
  //@ ensures \fresh(\result);
  public IndefiniteIterator iterator();
}
```

Figure 20: The interface `StrategyType`.

5.1.1 Strategies

In the following, we use the word "strategy" in the narrow, technical sense of the strategy pattern [24]. In particular, a *strategy object* is an object that has the interface given in Figure 20. A *strategy type* is a type that satisfies this interface. As can be seen in the interface declaration, a strategy object can be called upon to deliver an iterator that can provide test data. For example, the package `org.jmlspecs.jmlunit.strategies` has a class `StringStrategy`, which implements the `StrategyType` interface, and whose `iterator` method can deliver a new indefinite iterator object that supplies strings for testing.

The interface `StrategyType` is designed for providing test data of some reference type. For providing test data of primitive value types, the package `org.jmlspecs.jmlunit.strategies` has interfaces that provide the appropriately specialized indefinite iterator type. For example, the type `IntStrategyType` has a method, `intIterator` that returns a new `IntIterator`.

The main responsibilities of a strategy object's `iterator` method are to:

- construct an indefinite iterator that can return the test data, and
- to return a new, freshly allocated iterator each time the `iterator()` method is called.

Strategy types are an improvement over our earlier design for providing test data [15], because they allow users to avoid duplication of code that supplies test data.

```
protected IntStrategyType vintStrategy
  = new IntStrategy() {
      protected int[] addData() {
        return new int[] { 10, -22, 55, 3000 };
      }
    };
```

Figure 21: An example of providing test data of type `int`; this code is part of `Person_JML_TestData`.

```
public IntIterator vintIter(String methodName, int loopsThisSurrounds) {
  return vintStrategy.intIterator();
}
```

Figure 22: Implementation of `vintIter`, from the class `Person_JML_TestData`.

5.1.2 Choosing a Strategy

From the user's perspective, the choice of a strategy depends on the kind of test data to be provided, and on whether the user wishes to avoid observable aliasing among the data being used for testing.

Usually it is sensible to avoid observable aliasing among different test cases. If a mutable object is shared among different test cases, then the execution of one test can affect the starting condition of another test. That can make understanding the results of the second test difficult. Although such aliasing is actually desirable or necessary in some circumstances (such as when a singleton object is being used, e.g., as in a centralized log object), because of the potential for confusion it should not be entered into lightly. Thus the strategies discussed below try to avoid such aliasing. (We will, however, mention how such aliasing can be achieved if desired.)

On the other hand, when the objects involved are not mutable, observable aliasing is not possible, so sharing may be used freely. Similarly, when the test data is of a primitive value type, aliasing is not possible at all.

Once the question of whether to allow observable aliasing is settled, the user is left with a choice of strategies based on the kind of data to be supplied. We now describe these choices.

5.1.3 Strategies for Primitive Values

For Java's primitive value types, the package `org.jmlspecs.jmlunit.strategies` provides predefined and easily extensible strategies. For each primitive value type, such as `int` one can use one of two predefined strategies, such as `IntStrategy` or `IntBigStrategy`. The first of these, for example, `IntStrategy`, provides only a very small set of test data; for example, `IntStrategy` only provides 0, 1, and -1. `IntBigStrategy` is slightly more extensive, as it includes the maximum and minimum `int` values, and a few others. However, all these predefined strategies are intended to provide small sets of test data, since it is easy to enlarge the set of test data provided, but hard to make it smaller.

Users can add test data to one of the predefined strategies by defining a subclass that overrides the `addData()` method to provide the additional test data. The `addData()` method is overridden to return an array of objects. As shown in Figure 21, which extends the `IntStrategy` class, this can be easily accomplished by using an anonymous subclass and an array allocation expression, in the body of the method override.

To complete this example, the strategy can then be connected to the required interface method, as shown in Figure 22. The connection is accomplished by implementing the `vintIterator` method, and returning the result of calling the strategy's `intIterator` method.

The `jmlunit` tool automatically generates code that uses the smallest predefined strategy for a primitive value type. Comments in the generated code indicate how the user can extend the strategy,

```
protected StrategyType vStringStrategy
  = new StringStrategy() {
      protected Object[] addData() {
        return new String[] { "Baby", "Cortez", "Yoonsik" };
      }
    };

public IndefiniteIterator vStringIter
  (String methodName, int loopsThisSurrounds)
{
  return vStringStrategy.iterator();
}
```

Figure 23: An example of providing test data of type `String`, also from the class `Person_JML_TestData`.

if desired.

5.1.4 Strategies for Immutable Types

An immutable type is a type whose objects are immutable. Immutable objects can be treated in much the same way as primitive values, since sharing among immutable objects does not cause interference among tests.

The package `org.jmlspecs.jmlunit.strategies` provides an easily extensible strategy, the class `StringStrategy`, that can supply test data of type `String`. Again, `StringStrategy` only provides the bare minimum of test data: `null` and the empty string (`""`). Users would normally extend this strategy, which is extended in the same way as the strategies for primitive value types. For example, Figure 23 shows how to make a subclass of `StringStrategy` that provides three additional pieces of test data appropriate for testing the `Person` class.

For types with immutable objects for which no predefined strategy is provided, the user can subclass the class `ImmutableObjectAbstractStrategy`. This class provides the same extension mechanism as illustrated for the class `String`; indeed, the class `StringStrategy` is a subclass of `ImmutableObjectAbstractStrategy`.

The `jmlunit` tool has a list of reference types that it considers to be immutable. The type `java.lang.String` is treated as a special case, since it has predefined strategy, in the generated code. For other immutable types, the generated code uses `ImmutableObjectAbstractStrategy`, and indicates with comments how the user should extend the test data.

5.1.5 Supplying Cloneable Data

For types with mutable objects that have a publicly visible `clone()` method, one can make a subclass of the abstract strategy class `CloneableObjectAbstractStrategy`, and provide test data by overriding the `addData()` method, as described above. This strategy clones each object in the array of objects provided as it is returned by the `get()` method. By default the strategy only iterates over the `null` object.

Because Java does not provide a standard interface with a `clone()` method, the subclass of `CloneableObjectAbstractStrategy`, must also override the abstract `cloneElement(Object)` method, which is used by the `get()` method to clone each element of the array. However, the tool can generate a suitable override of the `cloneElement` method automatically, when the C`_JML_TestData` class is first created, so this is not difficult for users, but merely an annoyance of the Java type system.

```
protected StrategyType vPersonStrategy
  = new NewObjectAbstractStrategy() {
      protected Object make(int n) {
        switch (n) {
        case 0:
          return new Person("Baby");
        case 1:
          return new Person("Cortez");
        case 2:
          return new Person("Isabella");
        default:
          break;
        }
        throw new java.util.NoSuchElementException();
      }
    };

public IndefiniteIterator vPersonIter
  (String methodName, int loopsThisSurrounds)
{
  return vPersonStrategy.iterator();
}
```

Figure 24: An example of providing test data of type `Person`, again from the class `Person_JML_TestData`.

Having the tool treat types with a `clone()` method in this way greatly extends the number of types for which one can use the simple extension mechanism shown in the previous subsections. The idea is that, even if the test data is not immutable, side-effects on the clones will not affect each other, and so all the tests will stay independent of each other.

5.1.6 Supplying Mutable, Non-Cloneable Data

Some types have mutable objects but do not have a clone method. The class `Person` from our example is one such type. Supplying test data for such a type is slightly more complicated than for types with immutable or cloneable objects, since, to avoid observable aliasing one must freshly create an object each time the `get()` method is called. The abstract strategy for such types in the `org.jmlspecs.jmlunit.strategies` package is `NewObjectAbstractStrategy`. This strategy by default only provides `null` as test data.

The way that one extends `NewObjectAbstractStrategy` is to make a subclass that overrides the `make(int)` method to provide test data. This method uses its integer argument as an index to decide what object to create. An example is given in Figure 24, which uses anonymous subclass of the abstract new object strategy class. Note that, although `null` is included as one of the objects returned by this strategy, `null` is not used as a receiver, because the process of making test data filters out null receivers (see Section 4.3 above).

For types with mutable objects that are not cloneable, the `jmlunit` tool automatically generates a skeleton of the code like that in Figure 24.

Although the example and normal use of the `NewObjectAbstractStrategy` creates new objects to avoid observable aliasing among the test data, observable aliasing can be created if desired by not creating new objects each time the `make` method is called. One way to do this is to use a private field in the test data class, for example, an array of test objects of the appropriate type.

5.2 Discussion

Besides the strategies we have described above, we also have implemented some "higher-order" strategies in the package `org.jmlspecs.jmlunit.strategies`. The most straightforward of these are various kinds of composite [24] strategies, which can be used to sequence the test data provided by two or more strategies. There are also filtering strategies, which can filter test data based on some predicate (supplied as a method override).

Sometimes it is convenient to use test data of one type when building test data of another type. This can be accomplished by using the iterators of the other type in a loop when constructing data of the desired type.

As can be seen from the above examples, test data can be any object of the appropriate type. As shown in Figures 23 and 24, `null` is perfectly valid as test data for reference types. As might be expected, we have found that the use of `null` as a test input is very helpful in making sure the preconditions of methods that take reference types as parameters protect the method against null pointer exceptions.

The user must take care to either supply small amounts of test data for methods with many arguments or to limit the number of tests allowed in a test suite, since the number of test cases executed for a method is the product of each of the number of test items available for each argument. For example, with the test data access methods shown in Figures 24 and 21 the `addKgs` method is tested 21 times, since 21 is the product of the 3 non-null receivers and the 7 integers available for testing.[9]

The `org.jmlspecs.jmlunit.strategies` package has a predefined class, `LimitedTestSuite`, that can be used to limit the number of tests for a method. If an object of this type is returned by the `emptyTestSuiteFor` method (see Figure 9), then the number of tests for each method tested will be limited as specified by the argument to `LimitedTestSuite`'s constructor.

5.3 Running Test Cases

It is easy to perform test execution in our approach. This is done in the following steps.

1. Generate instrumented Java byte code for the type, C to be tested using the `jmlc` script; e.g., `jmlc -Q -U Person.java`.

2. Generate a JUnit test class, C`_JML_Test`, and, if necessary, a skeleton of the JUnit test case class, C`_JML_TestData`, using the `jmlunit` script; e.g., `jmlunit Person.java`.

3. Fill in the bodies of the test data access methods in the test data class, C`_JML_TestData`, to supply the test data.

4. Compile the user-defined test and test case classes using a Java compiler (other than `jmlc`); e.g., `javac Person_JML_Test*.java`.[10]

5. Run the test data class's tests, using the JML runtime assertion checking libraries, as provided by the `jml-junit` script or the `jmlrac` script; e.g., `jml-junit Person_JML_TestData` or `jmlrac Person_JML_TestData`.

5.3.1 Test Output

Figure 25 shows the result of running the test cases of Figure 19, i.e., the class `Person_JML_TestData`. It reveals the error that we mentioned in the caption of Figure 1.

[9]The 7 integers come from the 3 pieces of default data in `IntStrategy` and the 4 defined in the extension given in Figure 21.

[10]For this step, various JUnit and JML jar files have to be in the user's CLASSPATH.

```
% jmlrac Person_JML_TestData
.........F..F.....F..F.....F..F........
Time: 0.17
There were 6 failures:
1) addKgs(Person_JML_Test$TestAddKgs)junit.framework.AssertionFailedError:
      Method 'addKgs' applied to
      Receiver: Person("Baby",-1)
      Argument kgs: -1
Caused by: org.jmlspecs.jmlrac.runtime.JMLNormalPostconditionError:
    by method Person.addKgs regarding specifications at
File "Person.java", line 16, character 25 when
      '\old(weight + kgs)' is -1
      'kgs' is -1
      'this' is Person("Baby",-1)
      at Person.checkPost$addKgs$Person(Person.java:497)
      at Person.addKgs(Person.java:610)
      at Person_JML_Test$TestAddKgs.doCall(Person_JML_Test.java:263)
      at Person_JML_Test$OneTest.runTest(Person_JML_Test.java:85)
      at Person_JML_Test$OneTest.run(Person_JML_Test.java:75)
      at org.jmlspecs.jmlunit.JMLTestRunner.doRun(JMLTestRunner.java:170)
      at org.jmlspecs.jmlunit.JMLTestRunner.run(JMLTestRunner.java:145)
      at Person_JML_Test.main(Person_JML_Test.java:22)
2) addKgs(Person_JML_Test$TestAddKgs)junit.framework.AssertionFailedError:
      Method 'addKgs' applied to
      Receiver: Person("Baby",-22)
      Argument kgs: -22
Caused by: org.jmlspecs.jmlrac.runtime.JMLNormalPostconditionError:
    by method Person.addKgs regarding specifications at
File "Person.java", line 16, character 25 when
      '\old(weight + kgs)' is -22
      'kgs' is -22
      'this' is Person("Baby",-22)
      at Person.checkPost$addKgs$Person(Person.java:497)
      at Person.addKgs(Person.java:610)
      at Person_JML_Test$TestAddKgs.doCall(Person_JML_Test.java:263)
...

FAILURES!!!
Tests run: 33,  Failures: 6,  Errors: 0

JML test runs: 31/33 (meaningful/total)
```

Figure 25: Part of the output from running the tests in Person_JML_TestData. The output is truncated where ... appears near the end.

```
public void addKgs(int kgs) {
  if (kgs >= 0) {
    weight += kgs;
  } else {
    throw new IllegalArgumentException("Negative Kgs");
  }
}
```

Figure 26: Corrected implementation of method `addKgs` in class `Person`.

As the output in the figure shows, six tests failed for the method `addKgs`. For each failure, the test data that caused the failure is also printed. For example, in the first failure, the receiver, an object of class `Person` has name `Baby` and has had its weight changed to `-1`, due to the argument value of `-1` passed to `addKgs`. For the second failure, the argument was `-22`. More information is printed in the assertion violation message by the JML's runtime assertion checker, including the file name, line number, and column number where the violated assertion is located. From this information it is evident that the method returned normally, but violated its normal postcondition, since it is not supposed to return normally when the argument is negative. (It is supposed to throw an exception in that case.)

5.3.2 Fixing the Example

A corrected implementation of the method `addKgs` is shown in Figure 26. (Compare this with the specification and the faulty implementation shown in Figure 1.)

5.3.3 Counting Test Successes and Failures

To report the numbers of test successes or failures, the JUnit framework counts the number of tests run. However, the framework itself does not distinguish between meaningless tests and those that were successful. To get more accurate report, one can use our specialized test runner class, `JMLTestRunner`, instead of a JUnit's test runner class such as `junit.framework.textui.TestRunner`. (The `JMLTestRunner` is invoked automatically from the main method of the generated test case class, see Figure 19). The class `JMLTestRunner` reports the number of meaningful test runs and the total number of test runs in terms of test data, as shown in the last line of Figure 25. Such a report prevents the user from having a wrong impression that the class under test satisfied all tests when in fact no test has actually been executed due to all test cases being meaningless.[11]

Note that the total number of tests counted by JML differs from that counted by JUnit. The JML tests do not include the custom tests written by the user and the `test$isRACCompiled` test from Figure 18. In our example, we have no other custom tests, so the counts differ by just one.

6 Discussion

What should be the outcome of a test case if an invariant violation is detected even before the execution of the method body? Such a situation can arise if clients can directly write an object's fields, or if aliasing allows clients to manipulate the object's representation without calling its methods. The question is whether such invariant violations should be treated as a test failure or as a rejection of the test data (i.e., as a "meaningless" test). One reason for rejecting the test data is that one can consider the invariant to be part of the precondition. One may also consider an object malformed if it does not satisfy the invariant. However, treating such violations as if the test case

[11]We thank an anonymous referee of ECOOP 2002 for pointing out this problem.

were meaningless seems to mask the underlying violation of information hiding, and so our current implementation treats these as test failures.

7 Related Work

The traditional approach to implementing test oracles is to compare the result of a test execution with a user supplied, expected result [26, 42]. A test case, therefore, consists of a pair of input and output values. In our approach, however, a test case consists of only input values. And instead of comparing the actual and expected results, we observe if, for the given input values, the program under test satisfies the specified behavior. As a consequence, programmers are freed from not only the burden of writing test programs, often called *test drivers*, but also from the burden of pre-calculating presumably correct outputs and comparing them. The traditional schemes are constructive and direct whereas ours is behavior observing and indirect.

Several researchers have already noticed that if a program is formally specified, it should be possible to use the specification as an oracle [44, 47, 50]. Thus, the idea of automatically generating test oracles from formal specifications is not new, but the novelty lies in employing a runtime assertion checker as the test oracle engine. Peters and Parnas discussed their work on a tool that generates a test oracle from formal program documentation [44]. The behavior of program is specified in a relational program specification using tabular expressions, and the test oracle procedure, generated in C++, checks if an input and output pair satisfies the relation described by the specification. Their approach is limited to checking only pre and postconditions, thus allowing only a form of black-box tests. In our approach, however we also support *intra-conditions*, assertions that can be specified and checked within a method, i.e., on internal states [36]; thus our approach supports a form of white-box tests. Our approach also supports abstract value manipulation (see below) and object-oriented concepts such as specification inheritance.

Parasoft's commercial testing tool called Jtest [43] is perhaps most related to our work. The tool takes a similar approach as in ours in that it also uses pre- and postconditions as test oracles and relies on a runtime assertion checker to decide test successes or failures. The tool is also integrated with JUnit and, in addition, can automatically generate sample test cases from pre- and postconditions. However, due to its commercial nature, not much is published on its approach and implementation. One weak point of the Jtest tool, however, is that one cannot write pre and postconditions in terms of abstract values; assertions must be written in terms of concrete, implementation values or methods. In JML, one can manipulate abstract values (see below). This has an important consequence because one can write specifications for Java interfaces in a model-oriented style. Since an interface provides a perfect place to attach contracts between service providers and clients, it is paramount to be able to write specifications for interfaces. Without abstract value manipulation, however, one has to write interface specifications in terms of other methods declared in the interfaces, thus leading to algebraic-style specifications, which we found hard to Java programmers. Another advantage of our approach is that the iterator approach to supplying test data allows one to manipulate test cases in various ways, e.g., combining or filtering them.

Antoy and Hamlet describe an approach to checking the execution of an abstract data type's implementation against its specification [1]. Their approach is similar to the technique of multi-version programming [32] except that one version is an algebraic specification, serving as a test oracle. The algebraic specification is executed by (term) rewriting, and is compared with the execution of the implementation. For the comparison, the user has to provide an abstraction function that maps implementation states to abstract values. In our approach, no separate (specification) program needs to run in parallel with the implementation. Interestingly, however, the multi-version approach can be simulated to some extent in JML by using ghost variables. A *ghost variable* is a specification-only variable that is manipulated in specifications by using specification statements such as the `set` statement [36].

The Abstract State Machine Language (AsmL) [3] takes a similar approach of specification-based, multi-version programming. In AsmL, specifications are written as abstract programs that manipulate only abstract values. AsmL specifications are usually kept separate from program code, and thus they don't directly constrain program code. As in the approach of Antoy and Hamlet, specifications are normally executed separately from programs when doing runtime assertion checking, and violations are detected by comparing the two outputs [4]. This separation of specifications from programs is thus a paradigm shift from the so-called Design By Contract that our approach is based on.

Both the Jtest tool and ours are based on runtime assertion checking. There are now quite a few runtime assertion checking facilities developed and advocated by many different groups of researchers. One of the earliest and most popular approaches is Meyer's view of Design By Contract (DBC) embodied in the programming language Eiffel [38, 39, 40]. Eiffel's success in checking pre- and postconditions and encouraging the DBC discipline partly contributed to the availability of similar facilities in other programming languages, including C [48], C++ [19, 25, 46, 51], Java [5, 20, 23, 31, 34], .NET [2], Python [45], and Smalltalk [11]. These approaches vary widely from a simple assertion mechanism similar to the C `assert` macros, to full-fledged contract enforcement capabilities. Among all that we are aware of, however, none uses its assertion checking capability as a basis for automating program testing. Thus, the work of Jtest and ours are unique in the DBC community in using a runtime assertion checker to automate program testing. This aspect seems to be original and first explored independently in both the Jtest approach and ours.

Another difference between our work and other DBC work is that we use a formal specification language, JML, whose runtime assertion checker supports manipulation of abstract values. As far as we know, all other DBC tools work only with concrete program values. (Outside DBC, there are some specification languages that also support abstract values, notably the RESOLVE family [21, 22, 41] and the Abstract State Machine Language (AsmL) [3].) However, in JML, one can specify behavior in terms of abstract (specification) values, rather than concrete program values [9, 36, 37]. The so-called *model variables* — specification variables for holding not concrete program data but their abstractions — can be accompanied by `represents` clauses [16, 36]. A `represents` clause specifies an abstraction function (or relation) that maps concrete values into abstract values. This abstraction function is used by the runtime assertion checker to manipulate assertions written in terms of abstract values [13].

There are many research papers published on the subject of testing using formal specifications [8, 12, 17, 29, 33, 47, 49]. Most of these papers are concerned with techniques and methods for automatically generating test cases from formal specifications, though there are some addressing the problem of automatic generation of test oracles as noted before [44, 47, 50]. A general approach is to derive so-called *test conditions*, a description of test cases, from the formal specification of each program module [12]. The derived test conditions can be used to guide test selection and to measure comprehensiveness of an existing test suite, and sometimes they even can be turned into executable forms [12, 17]. The degree of support for automation varies widely from the derivation of test cases, to the actual test execution and even to the analysis of test results [17, 47]. Some approaches use existing specification languages [27, 29], and others have their own (specialized) languages for the description of test cases and test execution [12, 17, 47, 49]. All of these works are complementary to the approach described in this paper, since, except as noted above, they solve the problem of defining test cases which we do not attempt to solve, and they do not solve the problem of easing the task of writing test oracles, which we partially solve.

8 Conclusion and Future Work

We have presented a simple but effective approach to implementing test oracles from formal behavioral interface specifications. The idea is to use the runtime assertion checker as the decision procedure for test oracles. We have implemented this approach using JML, but other runtime assertion checkers can easily be adapted to work with our approach. There are two complications. The first is that the runtime assertion checker has to distinguish two kinds of precondition violations: those that arise from the call to a method and those that arise within the implementation of the method; the first kind of precondition violations is used to reject meaningless test cases, while the second indicates a test failure. The second is that the unit testing framework needs to distinguish three possible outcomes for test cases: a test execution can either be a success, a failure, or it can be meaningless.

Our approach trades the effort one might spend in writing code to construct expected test outputs for effort spent in writing formal specifications. Formal specifications are more concise and abstract than code, and hence we expect them to be more readable and maintainable. Formal specifications also serve as more readable documentation than testing code, and can be used as input to other tools such as extended static checkers [18].

Most testing methods do not check behavioral results, but focus only on defining what to test. Because most testing requires a large number of test cases, manually checking test results severely hampers its effectiveness, and makes repeated and frequent testing impractical. To remedy this, our approach automatically generates test oracles from formal specifications, and integrates these test oracles with a testing framework to automate test executions. This helps make our implementation practical. It also makes our approach a blend of formal verification and testing.

In sum, the main goal of our work —to ease the writing of testing code— has been achieved.

A main advantage of our approach is the improved automation of testing process, i.e., generation of test oracles from formal behavioral interface specifications and test executions. We expect that, due to the automation, writing test code will be easier. Indeed, this has been our experience. However, measuring this effect is future work.

Another advantage of our approach is that it helps to make formal methods more practical and usable in programming. One aspect of this is that test specifications and target programs can reside in the same file. We expect that this will have a positive effect in maintaining consistency between test specifications and the programs to be tested, although again this remains to be empirically verified.

A third advantage is that our approach can achieve the effect of both black-box testing and white-box testing. White-box testing can be achieved by specifying intra-conditions, predicates on internal states in addition to pre- and postconditions. Assertion facilities such as the `assert` statement are an example of intra conditions; they are widely used in programming and debugging. JML has several specification constructs for specifying intra-conditions.

Finally, in our approach a programmer may extend and add his own testing methods to the automatically generated test oracles. This can be done easily by adding hand-written test methods to a subclass of the automatically generated test class. Since iterators are used to supply test data, the programmer can also manipulate test cases in various ways, e.g., to combine, filter, etc.

Our approach frees the programmer from writing unit test code, but the programmer still has to supply actual test data by hand. In the future, we hope to partially alleviate this problem by automatically generating some of test inputs from the specifications. There are several approaches proposed by researchers to automatically deriving test cases from formal specifications. It would be very exciting to apply some of the published techniques to JML. JML has some features that may make this future work easier, in particular various forms of specification redundancy. In JML, a *redundant* part of a specification does not itself form part of the specification's contract, but instead is a formalized commentary on it [35]. One such feature are formalized examples, which can be thought of as specifying both test inputs and a description of the resulting post-state. However, for such formalized examples to be useful in generating test data, they would: (a) have to be specified constructively, and (b) it would have to be possible to invert the abstraction function, so as to build concrete representation values from them.

Another area of future work is to gain more experience with our approach. The application of our approach so far has been limited to the development of the JML support tools and examples that are shipped with JML, but our initial experience seems very promising. We were able to perform testing as an integral part of programming with minimal effort and to detect many kinds of errors. Almost half of the test failures that we encountered were caused by specification errors; this shows that our approach is useful for debugging specifications as well as code. However, we have yet to perform significant, empirical evaluation of the effectiveness of our approach.

JML and a version of the tool that implements our approach can be obtained through the JML web page at `http://www.jmlspecs.org`.

Acknowledgments

The work of both authors was supported in part by a grant from Electronics and Telecommunications Research Institute (ETRI) of South Korea, and by grants CCR-0097907 and CCR-0113181 from the US National Science Foundation. Thanks to Curtis Clifton, Markus Lumpe, Yoshiki Hayashi, and the ECOOP 2002 program committee for comments on earlier drafts of this paper. Thanks to David Cok for help with the `jmlunit` tool's implementation, and to David, Erik Poll, and others in the JML community for discussions on the `jmlunit` tool and its design.

References

[1] Sergio Antoy and Dick Hamlet. Automatically checking an implementation against its formal specification. *IEEE Transactions on Software Engineering*, 26(1):55–69, January 2000.

[2] Karine Arnout and Raphael Simon. The .NET contract wizard: Adding design by contract to languages other than Eiffel. In *Proceedings of TOOLS 39, 29 July -3 August 2001, Santa Barbara, California*, pages 14–23. IEEE Computer Society, 2001.

[3] Mike Barnett and Wolfram Schulte. The ABCs of specification: AsmL, behavior, and components. *Informatica*, 25(4):517–526, November 2001.

[4] Mike Barnett and Wolfram Schulte. Runtime verification of .NET contracts. *The Journal of Systems and Software*, 65(3):199–208, March 2003.

[5] D. Bartetzko, C. Fischer, M. Moller, and H. Wehrheim. Jass - Java with assertions. In *Workshop on Runtime Verification held in conjunction with the 13th Conference on Computer Aided Verification, CAV'01*, 2001.

[6] Kent Beck. *Extreme Programming Explained.* Addison-Wesley, 2000.

[7] Kent Beck and Erich Gamma. Test infected: Programmers love writing tests. *Java Report*, 3(7), July 1998.

[8] Gilles Bernot, Marie Claude Claudel, and Bruno Marre. Software testing based on formal specifications: a theory and a tool. *Software Engineering Journal*, 6(6):387–405, November 1991.

[9] Abhay Bhorkar. A run-time assertion checker for Java using JML. Technical Report TR #00-08, Department of Computer Science; Iowa State University, Ames, IA, May 2000.

[10] Lilian Burdy, Yoonsik Cheon, David Cok, Michael Ernst, Joe Kiniry, Gary T. Leavens, K. Rustan M. Leino, and Erik Poll. An overview of JML tools and applications. In Thomas Arts and Wan Fokkink, editors, *Eighth International Workshop on Formal Methods for Industrial Critical Systems (FMICS 03)*, volume 80 of *Electronic Notes in Theoretical Computer Science (ENTCS)*, pages 73–89. Elsevier, June 2003.

[11] Manuela Carrillo-Castellon, Jesus Garcia-Molina, Ernesto Pimentel, and Israel Repiso. Design by contract in Smalltalk. *Journal of Object-Oriented Programming*, 9(7):23–28, November/December 1996.

[12] Juei Chang, Debra J. Richardson, and Sriram Sankar. Structural specification-based testing with ADL. In *Proceedings of ISSTA 96, San Diego, CA*, pages 62–70. IEEE Computer Society, 1996.

[13] Yoonsik Cheon. A runtime assertion checker for the Java Modeling Language. Technical Report 03-09, Department of Computer Science, Iowa State University, Ames, IA, April 2003. The author's Ph.D. dissertation. Available from `archives.cs.iastate.edu`.

[14] Yoonsik Cheon and Gary T. Leavens. A runtime assertion checker for the Java Modeling Language (JML). In Hamid R. Arabnia and Youngsong Mun, editors, *Proceedings of the International Conference on Software Engineering Research and Practice (SERP '02), Las Vegas, Nevada, USA, June 24-27, 2002*, pages 322–328. CSREA Press, June 2002.

[15] Yoonsik Cheon and Gary T. Leavens. A simple and practical approach to unit testing: The JML and JUnit way. In Boris Magnusson, editor, *ECOOP 2002 — Object-Oriented Programming, 16th European Conference, Máalaga, Spain, Proceedings*, volume 2374 of *Lecture Notes in Computer Science*, pages 231–255, Berlin, June 2002. Springer-Verlag.

[16] Yoonsik Cheon, Gary T. Leavens, Murali Sitaraman, and Stephen Edwards. Model variables: Cleanly supporting abstraction in design by contract. Technical Report 03-10, Department of Computer Science, Iowa State University, April 2003. Available from `archives.cs.iastate.edu`.

[17] J. L. Crowley, J. F. Leathrum, and K. A. Liburdy. Isues in the full scale use of formal methods for automated testing. *ACM SIGSOFT Software Engineering Notes*, 21(3):71–78, May 1996.

[18] David L. Detlefs, K. Rustan M. Leino, Greg Nelson, and James B. Saxe. Extended static checking. SRC Research Report 159, Compaq Systems Research Center, 130 Lytton Ave., Palo Alto, Dec 1998.

[19] Carolyn K. Duby, Scott Meyers, and Steven P. Reiss. CCEL: A metalanguage for C++. In *USENIX C++ Technical Conference Proceedings*, pages 99–115, Portland, OR, August 1992. USENIX Assoc. Berkeley, CA, USA.

[20] Andrew Duncan and Urs Holzle. Adding contracts to Java with Handshake. Technical Report TRCS98-32, Department of Computer Science, University of California, Santa Barbara, CA, December 1998.

[21] Stephen H. Edwards, Wayne D. Heym, Timothy J. Long, Murali Sitaraman, and Bruce W. Weide. Part II: Specifying components in RESOLVE. *ACM SIGSOFT Software Engineering Notes*, 19(4):29–39, Oct 1994.

[22] Stephen H. Edwards, Gulam Shakir, Murali Sitaraman, Bruce W. Weide, and Joseph Hollingsworth. A framework for detecting interface violations in component-based software. In *Proceedings of the Fifth International Conference on Software Reuse*, pages 46–55. IEEE Computer Society Press, June 1998.

[23] Robert Bruce Findler and Matthias Felleisen. Behavioral interface contracts for Java. Technical Report CS TR00-366, Department of Computer Science, Rice University, Houston, TX, August 2000.

[24] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns, Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[25] Pedro Guerreiro. Simple support for design by contract in C++. In *Proceedings of TOOLS 39, 29 July -3 August 2001, Santa Barbara, California*, pages 24–34. IEEE Computer Society, 2001.

[26] R. G. Hamlet. Testing programs with the aid of a compiler. *IEEE Transactions on Software Engineering*, 3(4):279–290, July 1977.

[27] Teruo Higashino and Gregor v. Bochmann. Automatic analysis and test case derivation for a restricted class of LOTOS expressions with data parameters. *IEEE Transactions on Software Engineering*, 20(1):29–42, January 1994.

[28] Bart Jacobs and Eric Poll. A logic for the Java modeling language JML. In *Fundamental Approaches to Software Engineering (FASE'2001), Genova, Italy, 2001*, volume 2029 of *Lecture Notes in Computer Science*, pages 284–299. Springer-Verlag, 2001.

[29] Pankaj Jalote. Specification and testing of abstract data types. *Computing Languages*, 17(1):75–82, 1992.

[30] JUnit. Http://www.junit.org.

[31] Murat Karaorman, Urs Holzle, and John Bruno. jContractor: A reflective Java library to support design by contract. In Pierre Cointe, editor, *Meta-Level Architectures and Reflection, Second International Conference on Reflection '99, Saint-Malo, France, July 19–21, 1999, Proceedings*, volume 1616 of *Lecture Notes in Computer Science*, pages 175–196. Springer-Verlag, July 1999.

[32] John C. Knight and Nancy G. Leveson. An experimental evaluation of the assumption of independence in multiversion programming. *IEEE Transactions on Software Engineering*, 12(1):96–109, January 1986.

[33] Bogdan Korel and Ali M. Al-Yami. Automated regression test generation. In *Proceedings of ISSTA 98, Clearwater Beach, FL*, pages 143–152. IEEE Computer Society, 1998.

[34] Reto Kramer. iContract – the Java design by contract tool. *TOOLS 26: Technology of Object-Oriented Kanguages and Systems, Los Alamitos, California*, pages 295–307, 1998.

[35] Gary T. Leavens and Albert L. Baker. Enhancing the pre- and postcondition technique for more expressive specifications. In J. Davies J.M. Wing, J. Woodcock, editor, *FM'99 - Formal Methods, World Congress on Formal Methods in the Development of Computing Systems, Toulouse, France, September 1999. Proceedings, Volume II*, volume 1708 of *Lecture Notes in Computer Science*, pages 1087–1106. Springer-Verlag, September 1999.

[36] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. Preliminary design of JML: A behavioral interface specification language for Java. Technical Report 98-06p, Iowa State University, Department of Computer Science, August 2001. See `www.jmlspecs.org`.

[37] Gary T. Leavens, Albert L. Baker, and Clye Ruby. JML: A notation for detailed design. In Haim Kilov, Bernhard Rumpe, and Ian Simmonds, editors, *Behavioral Specifications of Businesses and Systems*, chapter 12, pages 175–188. Kluwer, 1999.

[38] B. Meyer. Applying design by contract. *IEEE Computer*, 25(10):40–51, October 1992.

[39] Bertrand Meyer. *Eiffel: The Language*. Object-Oriented Series. Prentice Hall, New York, NY, 1992.

[40] Bertrand Meyer. *Object-oriented Software Construction.* Prentice Hall, New York, NY, second edition, 1997.

[41] William F. Ogden, Murali Sitaraman, Bruce W. Weide, and Stuart H. Zweben. Part I: The RESOLVE framework and discipline — a research synopsis. *ACM SIGSOFT Software Engineering Notes*, 19(4):23–28, October 1994.

[42] D.J. Panzl. Automatic software test driver. *IEEE Computer*, pages 44–50, April 1978.

[43] Parasoft Corporation. Automatic Java^{TM} software and component testing: Using Jtest to automate unit testing and coding standard enforcement. Available from `http://www.parasoft.com/jsp/products/tech\_papers.jsp?product=Jtest`, as of Feb. 2003.

[44] Dennis Peters and David L. Parnas. Generating a test oracle from program documentation. In *Proceedings of ISSTA 94, Seattle, Washington, August, 1994*, pages 58–65. IEEE Computer Society, August 1994.

[45] Reinhold Plosch and Josef Pichler. Contracts: From analysis to C++ implementation. In *Proceedings of TOOLS 30*, pages 248–257. IEEE Computer Society, 1999.

[46] Sara Porat and Paul Fertig. Class assertions in C++. *Journal of Object-Oriented Programming*, 8(2):30–37, May 1995.

[47] Debra J. Richardson. TAOS: Testing with analysis and oracle support. In *Proceedings of ISSTA 94, Seattle, Washington, August, 1994*, pages 138–152. IEEE Computer Society, August 1994.

[48] David R. Rosenblum. A practical approach to programming with assertions. *IEEE Transactions on Software Engineering*, 21(1):19–31, January 1995.

[49] Sriram Sankar and Roger Hayes. ADL: An interface definition language for specifying and testing software. *ACM SIGPLAN Notices*, 29(8):13–21, August 1994. Proceedings of the Workshop on Interface Definition Language, Jeannette M. Wing (editor), Portland, Oregon.

[50] P. Stocks and D. Carrington. Test template framework: A specification-based test case study. In *Proceedings of the 1993 International Symposium on Software Testing and Analysis (ISSTA)*, pages 11–18. IEEE Computer Society, June 1993.

[51] David Welch and Scott Strong. An exception-based assertion mechanism for C++. *Journal of Object-Oriented Programming*, 11(4):50–60, July/August 1998.